(2) Making a hashtable of size n.

Insert the elements of the array to the hash table.

Now while searching if the no. stored does not match or if we get an empty box then we will return that index.

Psedo code : — Insert from array

— Iterate i = 1 to n+1

Search for i

— If Not present return i

else ~~return~~ continue.

```
int Find (A[..]) {
    Create_hash_table H.
    For ( i=0 to n-1)
    [ H[A[i]] = 1
    For (i= 1 to n+1)
    [ If (i ≠ H) return i
}
```

$\boxed{\text{Space \& Time } O(N)}$

(3) Rapin - Karp algorithm is a string searching algorithm.

It uses a Rolling hash to quickly filter out position of the text that cannot match the pattern. then checks for a match at the remaing positions.

S = |a|b|c|a|b|a|b|d| in blocks of 3 keeps rolling.

P = |a b a|

* The naive approach will be to compare P in S from starting to end char by char.

The time will be more $O((|S| - |P|) * |P|)$

$\downarrow$ Iterating. $\downarrow$ Comparing

*. The runtime can be bought down to liner using Rabin Karp. As shown above first calculating hash of all possible string by shifting the window

and matching will be done only if the hash matches with that of P.

Time complexity $= O\left(|S| + |P| + \text{no. of matches} \times \underline{|P|} \right)$

$|S|$ → Iterating over S

$|P|$ → For computing Hash

no. of matches → hash matches

$|P|$ → Comparing when hash matches.

The no. of matches

$\Rightarrow \dfrac{|S|}{|P|}$ (Assuming SUHA)

Time Complexity $= O\left(|S| + |T| + \dfrac{|S|}{|T|} \times |T| \right)$

$= O(|S| + |T|)$

If $P = P_1 P_2 P_3 \ldots P_{|T|}$ is a substring of S.

hashvalue (P) $=$ value $(P_1) \times \text{base}^{|T|}$ + value $(P_2) \times \text{base}^{|T|-1}$

... value $(P_{|T|})$

Value (P_1) → is the mapping value from char to int.

base → number of different char being mapped

For next Substring, hashvalue (NP)

$$= \left(\text{hash value } (P) - \text{Value } (P_1) \times base^{|T|} \right) \times base^{|T|} + \text{value } (NP.back()))$$

which is in O(1).

Function (S, P).

rolling hashT = 0.

ro

rolling hash = 0.

For i = 1 to |P|

rolling_hash *= base

rolling hash += value (P[i])

For i = 1 to |S| - |P|

If (rollinghash = rolling_hash T) check (S, P).

4) AVL

Minimum no. of nodes in a tree with heigh h can be represent as

$N(h) = N(h-1) + N(h-2) + 1 \quad h > 2$

$N(0) = 1$ &n $N(1) = 2$.

$\geq N(h-1) + N(h-2)$

$\geq N(h-2) + N(h-2)$

$\geq 2 * N(h-2)$.

$\geq 2 * 2 * N(h-4)$

$\vdots$

$\geq 2^{h/2} N(0)$

$n \geq N(h) \geq 2^{h/2}$

$n \geq 2^{h/2}$

$\log_2 n \geq h/2$

$2 \log_2 n \geq h$

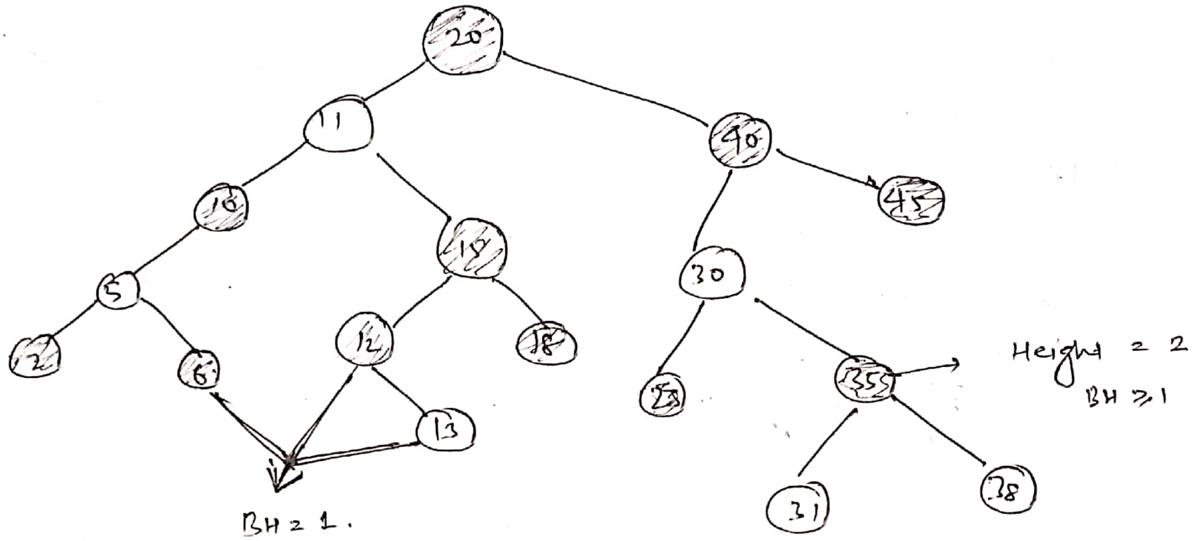
Height is then $O(\log n)$.

Rinam tree with property of

Red-Black tree : 2 things need to be proven.

① A subtree rooted at any node x has at least $2^{bh(x)} - 1$ internal nodes.

② Any node x with height $h(x)$ has $bh(x) \geq \frac{h(x)}{2}$.



BH = 1.

min no. of internal nodes = $2^1 - 1 = 1$

Statement ① → Using Induction.

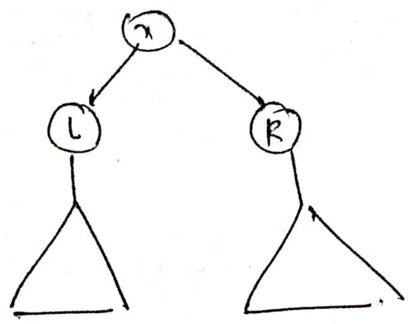
The base case is when $x = 0$ ie x is a leaf

according to the statement number of internal nodes $2^0 - 1 = 0$.

Since x is a leaf so this statement is true.

i.e. property of ① BST

Now let a node x with 2 children l and r



let $bh(x) = b$. Now if the colour of the child is red then its black height will also be b. However if the color of the child is black, then its black height will be b-1.

According to inductive hypothesis child must have at least

$2^{b-1} - 1 = 2^{bh(x)-1} - 1$ internal nodes.

We assumed inductive process to be true for child now we will show it true for parent i.e node x.

x must have at least 1 + least no. of internal nodes that can be present on the right child + least no. of internal nodes that can be present on the left child.

ie $2^{bh(x)-1} + 2^{bh(x)-1} + 1$

Internal node of $x \geq 2^{bh(x)-1} + 2^{bh(x)-1} + 1$

$x \geq 2 \times 2^{bh(x)-1} - 1$

$x \geq 2^{bh(x)} - 1$ Hence proved.

Coming to ② statement.

Since leaves are black and there can't be 2 consecutive red nodes, so the no. of red nodes can't exceed the no. of black nodes on any simple path from a node to a leaf. Therefore, we can say that at least half of the nodes on any simple path from root to a leaf, not including the node, must be black.

i.e $bh(x) \geq h(x)/2$.

using ① $n \geq 2^{bh(root)} - 1$

$n \geq 2^{h/2} - 1 \quad (bh(root) \geq h/2)$

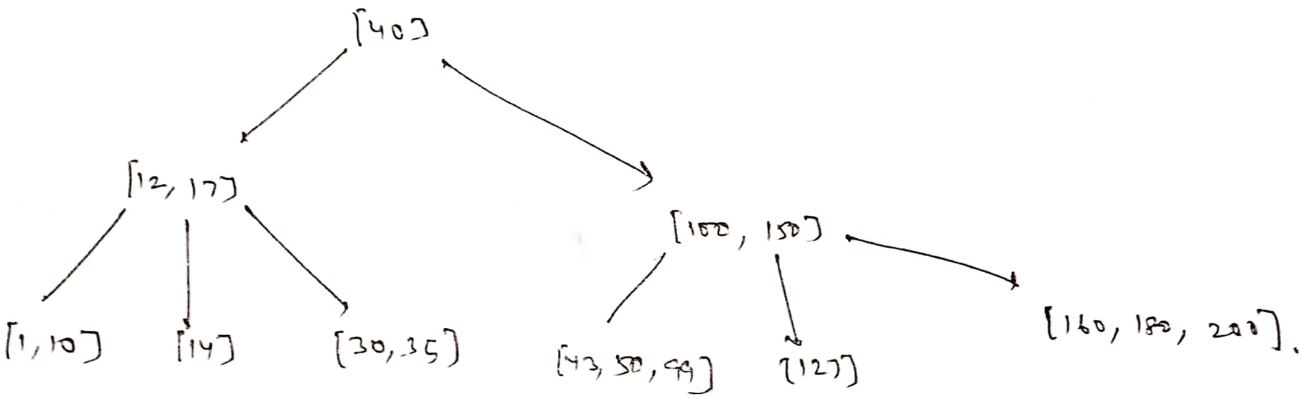
$n+1 \geq 2^{h/2}$. $\Rightarrow$ $\log(n+1) \geq h/2 \rightarrow h \leq 2\log(n+1)$

$\Rightarrow O(\log n)$

Scanned with CamScanner

5) 10, 12, 14, 100, 50, 40, 30, 1, 17, 150, 127, 200,

180, 99, 160, 43, 35.



b) Table Contraction

Table Doubling while items are inserted (Discussed in class)

Now if deletions are allowed then, As on deletion

the no. of elements decreases and as the result the

table-size also has to be reduced but as we

have done in expansion will not work in case of

Contraction. As when we will repetedly insert and

delete element this can sometime cost O(N) time.

For this to avoid we can do this when

no. of elements are $\frac{1}{4}$th of size which wont

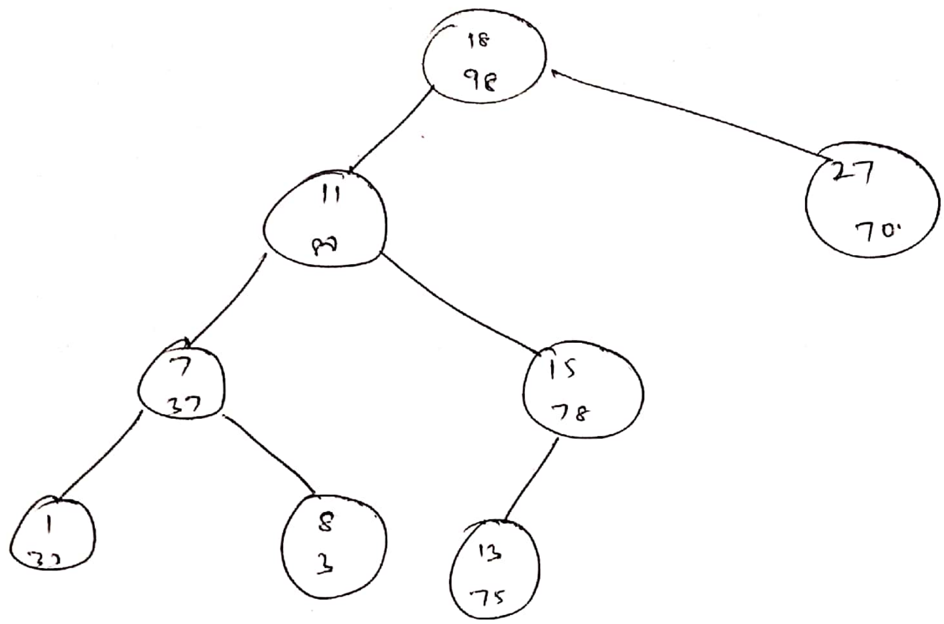
affect the above discussed case.

Scanned with CamScanner

7) Treap = Tree + heap.

Binary tree with property of ① BST

② Heap.



It can be seen we have 2 data in a node.

Hence it stores 2 data (x, y) which follow

BST property through x and Heap by y.

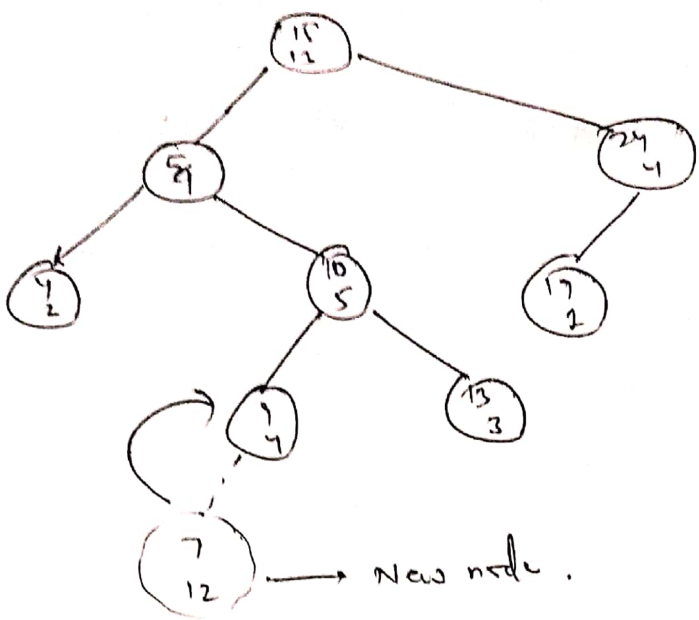
Operations : Insert (x, y) → $O(\log n)$

Search (x) → $O(\log n)$. (key x)

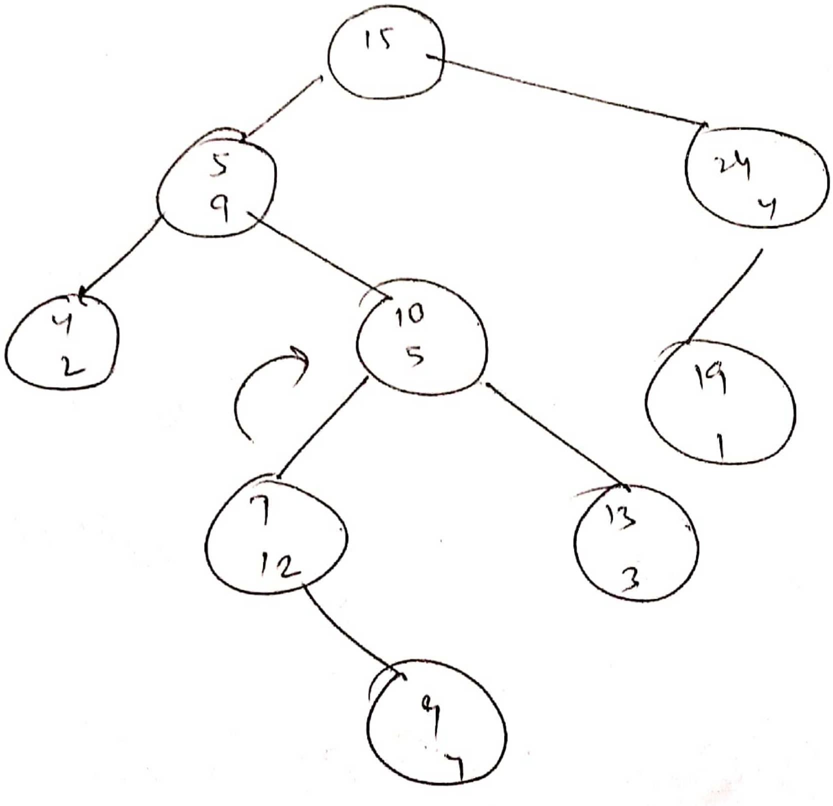
Build $([\,])$ → $O(\log n)$.

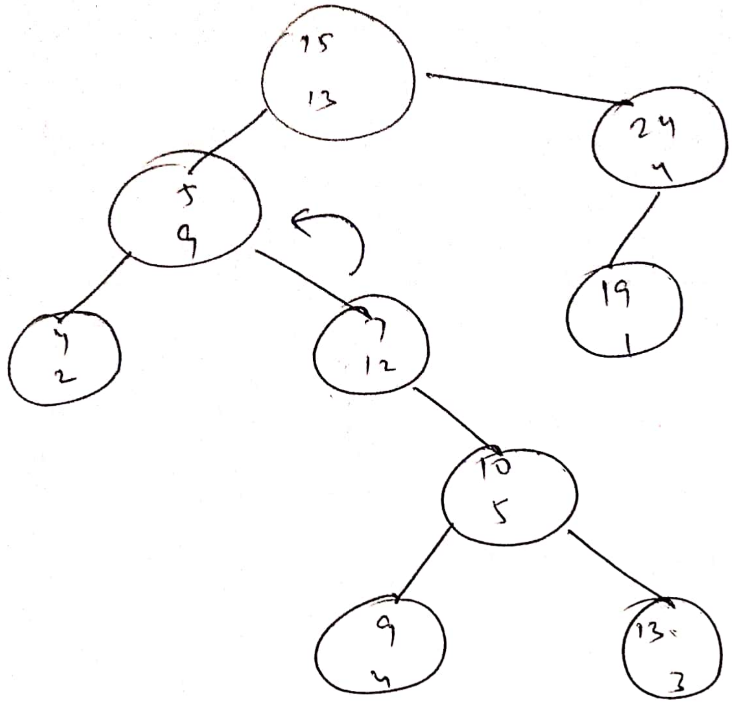
Delete (x) → $O(\log n)$

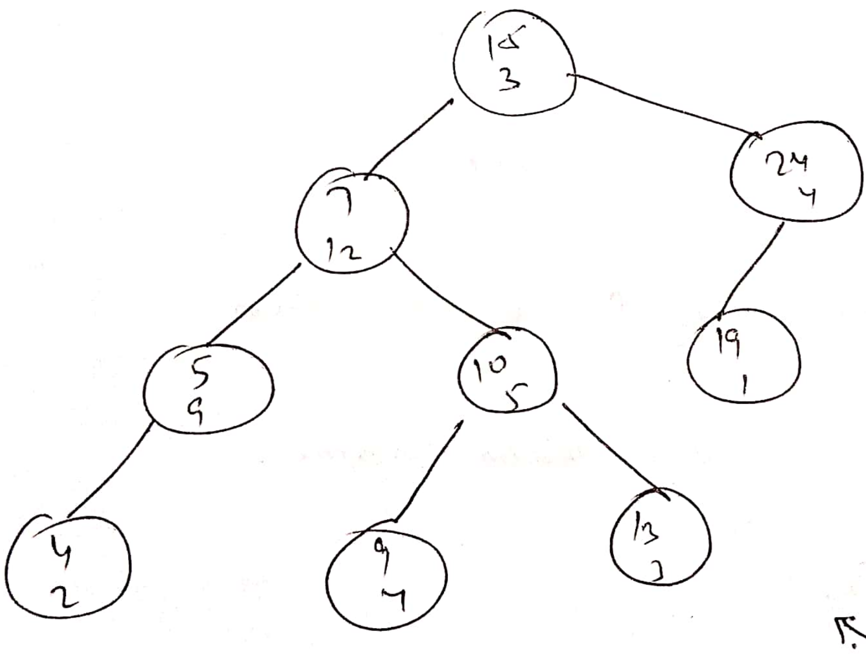
Example



Here BST is preserved but not heap.







After all rotations

(1)

Deletion is a problem when we have same mapped values of different data.

Case (I) If the ~~me~~ mapped value is found absent then this may return wrong for search.

Case (II) Deletion will be done for some other data.

We can use other data structure like a map or array which will tell if there is a collision at a position or not. We can then make them undeletable. Which will be then unability to delete few elements to avoid false negatives and false positives.

Scanned with CamScanner